Lynx: A Programmatic SAT Solver for the RNA-folding Problem

Vijay Ganesh and Charles W. O'Donnell
Mate Soos[†] and Srinivas Devadas and Martin Rinard and Armando Solar-Lezama

Massachusetts Institute of Technology, [†]Security Research Labs
{vganesh, cwo, devadas, rinard, asolar} @csail.mit.edu, [†]mate@srlabs.de

Abstract. This paper introduces Lynx, an incremental programmatic SAT solver that allows non-expert users to easily introduce domain-specific code into modern Conflict-driven Clause-learning (CDCL) SAT solvers, thus enabling users to control the behavior of the solver in ways not possible otherwise. The key idea of Lynx is an *easy-to-use callback interface* that enables non-expert users to *specialize the SAT solver to a class of Boolean instances* by allowing user-provided code to examine partial solutions generated by the solver during its search, and to dynamically and incrementally add CNF clauses back to the solver in response. While the power of incremental SAT solvers has been amply demonstrated in the SAT literature and in the context of DPLL(T), it has not been previously made available as a programmatic API that is easy to use for non-expert users. Lynx's callback interface is a simple yet very effective strategy that addresses this need.

We demonstrate the benefits of Lynx through a case-study from computational biology, namely, RNA secondary structure prediction. The constraints that make up this problem fall into two categories: structural constraints, which describe properties of the biological structure of the solution, and energetic constraints, which encode quantitative requirements that the solution must satisfy. We show that by introducing structural constraints on-demand through user provided code in Lynx, we can achieve, in comparison with standard SAT approaches, upto 30x reduction in the amount of memory required to obtain a solution and upto 100x reduction in solution time.

The callback interface can also be used as a template to expose other internals of a SAT solver such as branching heuristics or restart strategies, giving lay non-expert users more control over the solver. This interface has the potential to address an important problem with the search heuristics in modern CDCL SAT solvers, namely, their performance is unpredictable and non-uniform even for structurally similar Boolean formulas. The callback interface allows the user to compactly specify additional information about an input instance that can nudge the solver's heuristics in the right direction, i.e., makes the solver more efficient and predictable.

1 Introduction

CDCL (Conflict-Driven Clause Learning) Boolean SAT solvers have had a huge impact on a variety of domains ranging from program analysis to AI [3]. This success can be attributed to dramatic improvement in their efficiency in recent years thanks largely to techniques such as clause learning, effective branching heuristics and restart strategies [3].

Unfortunately, there are many important domains where *simple standard translations* of problems from these domains to Boolean formulas lead to representations that are either too large or too difficult-to-solve for even the best of solvers. However, users are often aware of non-standard translation strategies and/or heuristics that can enable a cost-effective SAT-based solution to such problems. Furthermore, it is not known whether there is a small set of non-standard strategies and heuristics that will work well for all problems and which can be easily coded by the SAT solver developer. On the contrary, the evidence suggests that for many such problems the translation strategies and heuristics that work well are specific to the domain. A developer cannot be expected to know all such strategies and code them up into a single solver.

Consequently, it is important to enable users in such a way that they can specialize and adapt SAT solver heuristics to their problem domains. Ideally, the user should be able to adapt the solver with minimal effort and without breaking subtle solver-code invariants. The resultant specialized solver should be adaptive, efficient, easy to build and maintain. Equally importantly, users should not be burdened with knowing too much about SAT solvers and related technologies.

Our Contributions

- To address the above-described problem, we provide a simple yet effective mechanism through a callback interface (aka API) in our solver Lynx which allows user-specified code to examine partial solutions generated by the SAT solver, and add CNF clauses back to the solver in response. The user code is called inside the inner loop of the SAT solver (we used CryptoMiniSat [26]). Thus, users can implement non-standard translations or heuristics tightly integrated into the solver and build highly adaptable solvers specialized for their specific problem domains.

  We call such solvers *programmatic*, i.e., the user can programmatically influence solver behavior and adapt it to their specific problem domain in ways that is difficult to acheive otherwise. Programmatic solvers address the ”solvers-are-unpredictable-blackboxes” problem by giving users more control over their search heuristics.

  Lynx’s interface is simple enough that non-expert users can easily modify and maintain their code. Furthermore, unlike related approaches discussed below, our approach places very few restrictions on user-code and separates user code from SAT solver cleanly. Consequently, users need not worry about breaking any subtle invariants of the base SAT solver.
- Using Lynx we developed the first SAT-based tool for solving the RNA-folding prediction problem. We present a detailed experimental evaluation of our technique in comparison with standard approaches. We use the above-mentioned callback interface in Lynx to incrementally translate the RNA-folding structure prediction problem into Boolean formulas. The interface allows us do the translation in an online incremental fashion inside the inner loop of the SAT solver, thus allowing a tighter, highly targeted and more efficient integration of the SAT solver and the translator.

Previous Approaches

Incremental solvers have been proposed as a solution to the above-mentioned issue of *simple but inefficient standard translations* from many problem domains to Boolean SAT. Instead of translating the entire problem-instance into a potentially very large Boolean formula in one step, an incremental solver uses *abstraction-refinement approaches* to translate the instance into a Boolean formula incrementally and call the solver on these abstractions, and refining the abstractions as necessary to get the correct result to the satisfiability question for the input problem-instance [3]. Typically these translations are done by a translation layer outside the SAT solver.

Such incremental SAT solvers with an outside abstraction-refinement loop are relatively easy to build. However, the problem with such an approach is that it may not be the most efficient for the problem-at-hand. Indeed, Ohrimenko et al. [21] have proposed incremental translation of problems to SAT where the integration of the solver and the incremental translation is much tighter and more efficient than an outer layer translator. However, their implementation is non-adaptive, and is specific to a class of difference logic formulas. In other words, they haven't provided an API that users can use to easily adapt or extend the solver for a class of Boolean formulas (e.g., users could use such an API to make the solver incremental, where the integration of the user's incremental translator with the solver is very tight in a sense that we make specific below).

An example of an API that allows users to adapt or extend solvers is the powerful idea of DPLL(T) [13] aimed at solving Boolean combination of formulas in rich theories such as integer linear arithmetic, uninterpreted functions and datatypes (aka SMT solvers [3]). In this approach, there is a tight integration of a CDCL SAT solver with a so-called *theory solver* (aka a T-solver) that can handle conjunction of constraints represented in a rich logic. The CDCL SAT solver does the search on the Boolean structure of the formula without knowing the semantics of the literals, while the T-solver reasons about the literals themselves adding any new derived literals back to the Boolean CDCL solver appropriately. The tight integration enables the T-solver to influence the CDCL solver's behavior in ways not possible otherwise, and the resultant combination is typically a very powerful solver than can handle arbitrary Boolean combination of theory formulas efficiently.

A lay non-expert user could implement a "T-solver" using the DPLL(T) framework that reasons about a specific domain (say, theory of RNA folding) and adds constraints incrementally to the SAT solver. The resultant combination can be tight and incremental ala Ohrimenko and Stuckey et al. [21]. However, while DPLL(T) is a very powerful idea and helps create domain-specific solvers, it does impose strict requirements on the user-specified code (T-solver) in order to ensure that the resultant combination is sound and complete (This is a well-understood and accepted fact in the SAT/SMT solver community). Such requirements make perfect sense for constructing powerful SMT solvers, the problem for which the DPLL(T) approach was invented by Oliveras, Nieuwenhuis, and independently by Tinelli et al [13]. However, for the lay non-expert users such requirements may be difficult to satisfy, and often not essential. A typical biologist is more concerned about solver efficiency and correctness for a class of problem-instances, and probably cares less about completeness of a solver with respect to all instances. Furthermore, such users may be willing to sacrifice completeness as long as the resultant combination of user-code and solver is efficient, and the user can easily experiment with and modify their code to get the correct solution.

RNA-folding with Lynx

To explore the benefits of using the callback interface provided by Lynx, we applied the technique to the problem of RNA folding. This is an application of significant practical relevance: understanding RNA folding is crucial to understanding a number of biological processes, including the replication of single-strand RNA viruses such as the poliovirus which causes polio in humans. Moreover, RNA prediction actually shares important similarities with other structure prediction problems of biological interest. This problem is a particularly suitable benchmark for our approach. First, a SAT based solution to this problem would be particularly desirable because it would give researchers the ability to easily experiment with different formulations for the basic problem. Moreover, previous work in the literature has succeeded in formalizing the problem in a form that lends itself very naturally to solution with a Boolean SAT solver. SAT solver-based solutions, however, have been elusive because the standard encoding leads to Boolean SAT instances that are too big for modern solvers to handle. Using Lynx's callback interface allowed us to encode instances of the RNA folding problem in a memory efficient manner, thus produce the first successful SAT-based solution to this problem. The resultant incremental (or online abstraction-refinement) solver led to a 30-fold reduction in the amount of memory required to solve some of these problems compared to standard SAT approach, and demonstrated dramatic time improvements over standard abstraction refinement techniques.

Paper Layout: The layout of the rest of the paper is as follows: In Section 2 we provide a detailed overview of the callback interface and discuss the implementation. In Section 3 we provide a self-contained description of the RNA-folding structure prediction problem. In Section 4 we provide detailed description of our experimental setup and results. We review the related work in Section 5, and conclude in Section 6.

2 Incrementality in Lynx

This section describes the details of how the callback interface in Lynx allows us to make the solver incremental, what we sometimes also refer to as online abstraction-refinement or OAR. In order to facilitate the description, we introduce a simple running example which shares some of the features of the more complex biology application.

The running example is a formula of the form $P(\boldsymbol{x}) \wedge C(\boldsymbol{x})$ over a vector $\boldsymbol{x} = \langle x_0, x_1, \ldots, x_N \rangle$ of Boolean variables, where $P(\boldsymbol{x})$ consists of some arbitrary set of constraints and $C(\boldsymbol{x})$ is a cardinality constraint that says that no more than 2 bits in $\boldsymbol{x}$ can be set to 1.

$$C(\boldsymbol{x}) \equiv \forall_{i \neq j \neq k} (\neg x_i \vee \neg x_j \vee \neg x_k)$$

The above definition of $C(\boldsymbol{x})$ can be trivially encoded as a set of N^3 CNF clauses—too many for large values of N. For this specific case, more efficient encodings exist that use only $O(N)$ clauses, but they are more complicated and require the introduction of additional SAT variables. By contrast, online abstraction refinement will allow us to use the simple encoding without having to pay the price of introducing N^3 clauses.

The first step in using OAR is to divide the problem into a core set of clauses which will be added to the solver from the very beginning, and a different set of *dynamic* clauses that will be added to the solver incrementally by a *callback function*. The callback function is a function provided by the solver user $\mathcal{M}$ that produces a set of clauses given a partial assignment to the variables. A partial assignment sets each variable in the problem to either 1, 0, or $\perp$ (undefined), and is represented as a vector $\boldsymbol{t} \in \{0, 1, \perp\}^N$.

In the case of the example, we define $P(\boldsymbol{x})$ to be the core clauses, and $C(\boldsymbol{x})$ to be the clauses added dynamically by a callback function defined by the following:

$$\mathcal{M}(\boldsymbol{t}) \equiv \{(\neg x_i \vee \neg x_j \vee \neg x_k) \mid i \neq j \neq k \wedge t_i = t_j = t_k = 1\}$$

This callback function receives a partial assignment $\boldsymbol{t}$, and returns a set of clauses of the form $(\neg x_i \vee \neg x_j \vee \neg x_k)$ where x_i, x_j and x_k are variables set to 1 in the partial assignment (i.e., $t_i = t_j = t_k = 1$). The clauses produced by the callback function eliminate those incorrect solutions that would have been eliminated by $C(\boldsymbol{x})$, so running the solver with constraints $P(\boldsymbol{x})$ and callback function $\mathcal{M}$ is the same as solving $P(\boldsymbol{x}) \wedge C(\boldsymbol{x})$.

Lynx incorporates the callback function into the solution process by invoking it periodically with the current partial assignment. If the callback function returns any clauses, these are incorporated into the problem, and the process continues until an assignment is found that: a) satisfies all the core constraints, b) satisfies all the constraints ever produced by the callback function, and c) produces an empty set of clauses when the callback function is applied to it. If the input problem is unsatisfiable, the solver with the callback function is guaranteed to report unsatisfiable and terminate. In the case of our running example, the guarantees provided by the solver allow us to establish that the solution provided by Lynx is a correct solution to the initial set of constraints. In general, these guarantees allow us to make formal claims about the relationship between the solutions produced by Lynx and the space of solutions to the original problem.

2.1 Anatomy of a callback function

In Lynx a callback function is a fragment of code that checks a partial assignment to the variables and produces clauses in response. The callback function can perform arbitrary computation, but is limited in terms of the interactions that it can have with the solver. These limitations are intentional and they are designed to prevent the callback function code from tampering directly with the highly sensitive search heuristics of the solver, and free the user from worrying about breaking low-level subtle solver invariants. The callback function interacts with the solver through the following functions:

- **varIter()** returns an iterator over all the important variables that have values so far, sorted in the order in which they were assigned. Users define what variables in the program are important to the callback function, allowing the callback function to ignore assignments to other variables (aka the assignment stack).
- **recentVarIter()** returns an iterator over all the variables that have been assigned since the last time the callback function executed.
- **addClause(Clause c)** allows the callback function to add a clause to the current problem.

In writing callback functions, users must balance the elimination power of the callback function with execution efficiency. For our running example, a simple and efficient callback function is as follows:

```
 Iterator  fst = recentVarIter ();
while( fst .hasNext())
{  Lit l1 = fst .next ();
   if (! l1 . sign ())
   {   Iterator  sec = varIter (); Lit temp = 0;
       while(sec .hasNext())
       {  Lit l2 = sec .hasNext();
          if (! l2 . sign ())
          {    if (temp!=0) addClause({ ¬ l1, ¬l2, ¬temp});
```

```
                temp = l2;}
    }  } }
```

This callback function produces only a subset of the clauses produced by $\mathcal{M}$ presented earlier, so it eliminates fewer bad assignments; the tradeoff is that it is efficient to compute. Further, its outer loop uses the iterator returned by recentVarIter (). This prevents the callback function from doing a lot of repeated work from one invocation to the next, dramatically reducing its overhead.

2.2 Implementation

We implemented Lynx on top of CryptoMiniSat [26] with a handful of simple modifications. CryptoMiniSat is a derivative of MiniSat [11], with various advanced features such as clause subsumption and strengthening. Like all DPLL-style SAT [9] solvers, CryptoMiniSat uses the main loop:

```
     while not done{
1:       Conflict c = propagate ();
         if (c != null)
2:           /* handle conflict and backtrack */
         else
3:           /* decide on a new variable to enqueue, or quit when done*/
     }
```

We modified this loop to call the callback function before deciding on a new variable to enqueue in line 3. This way, the callback function is called periodically and the system cannot return a satisfying assignment without checking the callback function first. If the callback function returns any clauses, these are preprocessed and added to the clause database, and the solver backtracks using the same strategy it uses for conflict clauses. The implementation involved a number of low-level detailes, but required minimal modifications to CryptoMiniSat, and could be easily implemented in other SAT solvers such as MiniSat [11].

3 Biological Problem Overview

RNA is a versatile polymer essential all of life. A chain of covalently bound nucleotides, RNA classically acts as a cellular messenger which duplicates DNA sequence information in the nucleus/nucleoid and transports that code to ribosomes for the construction of proteins. However, this chain can also fold in on itself into a 3-dimensional globular molecule which catalyzes biological reactions by itself. In fact, modern studies have suggested that such non-coding RNA (ncRNA) may play even a bigger cellular role than messenger RNA, with significant effects on metabolism, signal transduction, gene regulation, and chromosome inactivation. Such RNA function is determined by its nucleotide composition and 3-dimensional structure, however, relatively little ncRNA structural data is known [**?**], severely limited our understanding of these mechanisms. Therefore, algorithmic prediction of RNA structure from its nucleotide sequence has been a longstanding computational goal, and has become an increasingly important biological tool as ncRNA's are implicated in more and more processes.

3.1 Structure prediction via SAT

In this paper, the computational problem we address on is how to correctly attribute a unique structural state to each nucleic acid (or groups of nucleic acids) within an RNA polymer sequence. This problem has a long history of solutions based on many different algorithmic models — the most successful of which using a recursive, grammatical approach introduced by Zuker [28]. In this biophysical model, each nucleotide is allowed to form a pairwise bond with another, and each pair is assigned an energetic cost based on spatially adjacent nucleotide types [19]. The most likely structure is predicted by optimizing pairing configuration according to a fixed thermodynamic scoring system (energy minimization). Efficient computation is made possible through of the imposition of specific, often biologically-inspired model restrictions — for example, limiting base-pairs to be sequentially nested (i.e. no "pseudoknots") and scoring only a subset of all potential energetic interactions (i.e. only Watson-Crick or wobble base-pairs). Unfortunately, this entangles the optimization techniques used with a particular set of biological assumptions. While these methods have shown good predictive accuracy, changes to the algorithm can be difficult to implement as new scientific data comes to light. For example, it has been shown that a more complex description of the RNA interaction energetics can lead to greatly improved results [22].

We propose a declarative approach for the structure prediction problem, providing a decoupled platform for reasoning about biological concepts in clear, succinct rules, backed by the powerful generic optimization of CDCL SAT solvers. This allows biological models to be tested and flexibly refined using a constraint-based philosophy, independent of performance improvements to the underlying solver. To study this approach, we have implemented an RNA structure prediction algorithm using Lynx. Rather than compare the benefits and disadvantages of different biological models, we base our implementation on an RNA scoring model recently proposed by Kato, et al. for integer programming optimization [23]. Indeed, other models outperform this scoring system's accuracy, but we believe our results are easily generalizable to greater classes of RNA structures [4] and more complex (non-RNA) structure prediction problems in general. To implement energy minimization as a SAT-based decision procedure, we pose the question of whether an assignment exists that is lower than a certain energy threshold and perform iterative binary search. Despite this search routine, this approach can often be more efficient than the dynamic programming methods used by grammatical models as the problem can be finely partitioned into smaller jobs that are run in parallel. Further, when a sub-optimal solution is sufficient, this method quickly short-circuits, along with a guarantee of how near the solution is to optimality. For clarity, timing results in this paper reflect a single threaded search, but we have otherwise implemented parallelization techniques that work across multiple cores and machines.

3.2 RNA secondary structure prediction with pseudoknots

We note that the RNA prediction algorithm described here differentiates itself from classical prediction methods in its goal of predicting pseudoknots. Earlier grammar-based predictors allowed only base-pairs to occur in a recursively nested fashion (i.e. for every base-pair i-j there exists no base-pair k-l such that $i < k < j < l$) to enable highly efficient energy minimization via dynamic programming. However, pseudo-knotted structures which break this restriction are known to be essential to a number of functions, such as the Diels-Alder ribozyme and mouse mammary tumor virus [27]. However, predicting pseudoknotted structures is computationally a much harder problem with fewer solutions [20, 25, 23, 24]. In fact, the prediction of truly arbitrary pseudoknots has been shown NP-complete [17], and classes of pseudoknotted structures are often more easily defined by the algorithms which recognize them rather than their biological significance [8]. This motivates the use of a declarative approach, which allows on to easily explore different trade-offs between representation and optimization, especially if the underlying scoring system is changed from the standard Watson-Crick/wobble base-pair models to more complex interactions [22]. However, in the remainder of this work we again restrict ourselves to the model proposed by Kato, et al. [23].

3.3 Encoding RNA structure prediction in SAT

Our SAT encoding is formulated by two sets of constraints, structural and energetic, that control the assignment of a vector of free variables which represent the final structural solution. The assignment of each free variable indicates whether two nucleotides are base-paired in the final RNA structure, fixed by structural constraints and an associated energetic score. Figure 1 depicts this formulation.

Solution variables: The set of all properly-nested base-pairs within the final output RNA structure is represented by the variables $X_{i,j}$: where i and j indicate the sequence position of two nucleotides, a value $X_{i,j} = \mathbf{T}$ indicates a hydrogen bond base-pair exists between nucleotides at i and j, and $X_{i,j} = \mathbf{F}$ indicates that no base-pairing occurs between positions i and j. The set of pseudoknotted base pairs that cannot be properly nested are similarly represented by the independent variables $Y_{i,j}$. In this way pseudoknots are represented solely by the alignment of properly-nested $X_{i,j}$ pairs and properly-nested $Y_{i,j}$ pairs. Since RNA structure permits any nucleotide position i to pair with any other position j, a valid biological structure requires a complete assignment of all $X_{i,j}$s and $Y_{i,j}$s for every i, j ($0 \leq i, j < length(sequence)$). Therefore, the number of solution variables, the number of resultant constraints, and thus the difficulty of the SAT problem depends directly on the sequence length of the input RNA.

Structural constraints: The structural representation places requirements on the assignment of the solution bits $X_{i,j}$ and $Y_{i,j}$ to ensure a biologically consistent structure. Therefore, we declare the following constraints, which must be satisfied in any valid solution:

- Every position i can at most pair with one other position j, independent of whether that pairing is properly-nested or a pseudoknot (Figure 1(a-d)). Four straightforward constraints ensure this:

$$\begin{gathered} \forall i, j, k, \quad i < j < k \\ (X_{i,j} \wedge X_{j,k}) = \mathbf{F} \wedge (Y_{i,j} \wedge Y_{j,k}) = \mathbf{F} \quad \wedge \\ (X_{i,j} \wedge Y_{j,k}) = \mathbf{F} \wedge (Y_{i,j} \wedge X_{j,k}) = \mathbf{F} \end{gathered}$$

- All base-pairs i, j are properly nested or a pseudoknot, but not both (Figure 1(e)):

$$\forall i, j \ \ (X_{i,j} \wedge Y_{i,j}) = \mathbf{F}$$

- We define all $X_{i,j}$ and $Y_{i,j}$ base-pairs to be independently knot-free (Figure 1(f-g)):

$$\begin{gathered} \forall i, j, k, l \quad s.t. \ i < k < j < l \\ (X_{i,j} \wedge X_{k,l}) = \mathbf{F} \wedge (Y_{i,j} \wedge Y_{k,l}) = \mathbf{F} \end{gathered}$$

- We only permit bifurcations within the “normal” base-pairs in $X_{i,j}$ since pseudoknots are rare and deserve distinct energetic treatment. Therefore (Figure 1(h):

$$\forall i,j,k,l,\ i<k<j<l \quad (Y_{i,j} \wedge Y_{k,l}) = \mathbf{F}$$

- Finally, the class of structures with “double-crossing” pseudoknots are rare and present unusual energetics which are not handled by the energy model we use, thus we constrain pseudoknots to only cross at most once (Figure 1(i-j)):

$$\begin{gathered}\forall i,j,k,l,m,n, \qquad i<m<j<k<n<l \\ (X_{i,j} \wedge Y_{m,n}) \implies (X_{k,l} = \mathbf{F}) \quad \wedge \\ (X_{k,l} \wedge Y_{m,n}) \implies (X_{i,j} = \mathbf{F})\end{gathered}$$

Energetic constraints: The total energy of an RNA structure is defined as the sum of experimentally-derived energy parameters [28, 23] for every constituent base-pair *stack*, where a stack indicates two adjacent base pairs, e.g. $X_{i,j}$ and $X_{i+1,j-1}$. Energy parameters are given in terms of base-pair stacks because nucleotide π-orbital overlap serves as a dominant stabilizing factor in RNA structure. Thus, an energy value is assigned to every base-pair stack $X_{i,j}X_{i+1,j-1}$ according to the four nucleotide types at sequence positions i, j, $i+1$, and $j-1$ (Parameters found in [23]). By including a logical adder of all possible energetic assignments, we can then define a valid solution as an assignment of $X_{i,j}$ and $Y_{i,j}$ (subject to structural constraints), where the output of the adder overcomes some minimum threshold energy $E_{threshold}$ (the energy bound). As a logical declaration, we write:

$$\begin{aligned}\forall i,j,\ i<j\ (X_{i,j} \wedge X_{i+1,j-1}) = \mathbf{T} &\Rightarrow (E_{X_{i,j}} = EnergyConstant_{(i,j,i+1,j-1)}) \quad \wedge \\ (Y_{i,j} \wedge Y_{i+1,j-1}) = \mathbf{T} &\Rightarrow (E_{Y_{i,j}} = EnergyConstant_{(i,j,i+1,j-1)}) \quad \wedge \\ (X_{i,j} \wedge X_{i+1,j-1}) = \mathbf{F} &\Rightarrow (E_{X_{i,j}} = 0) \quad \wedge \\ (Y_{i,j} \wedge Y_{i+1,j-1}) = \mathbf{F} &\Rightarrow (E_{Y_{i,j}} = 0),\end{aligned}$$

where $EnergyConstant(i,j,i+1,j-1)$ indicates the energy score of the four nucleotides found at positions i, j, $i+1$, and $j+1$ base-pairing and stacking, and

$$\sum_{\forall i,j} (E_{X_{i,j}} + E_{Y_{i,j}}) \geq E_{threshold}.$$

Finally, to enforce that all assigned base-pairs are accounted within the adder by stacking energy parameters, we require:

$$\begin{gathered}\forall i,j \quad s.t.\ i<j \\ (\overline{X_{i-1,j+1}} \wedge X_{i,j} \wedge \overline{X_{i+1,j-1}}) = \mathbf{F} \quad \wedge \\ (\overline{Y_{i-1,j+1}} \wedge Y_{i,j} \wedge \overline{Y_{i+1,j-1}}) = \mathbf{F}\end{gathered}$$

4 Experimental Results

In this section we describe the results of our experimental evaluation of Lynx and competing approaches over input tests obtained from a set of RNA sequences. As described in detail in 3, we solve the two dimensional RNA optimum structure prediction problem (where the structures may have pseudoknots). We ran all experiments on a 3GHz Intel Xeon X5460 with 64GB of RAM and a 6MB L2 cache with 1 hour timeout per SAT instance.

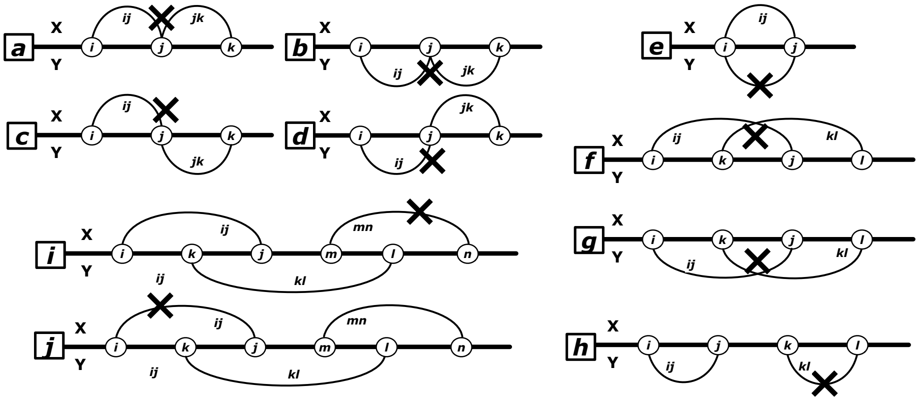

Fig. 1. RNA Constraints

4.1 Description of Input Tests

We acquired a set of benchmark RNA sequences and structures from the PseudoBase website [1]. These RNA sequences are widely used by computational biologists for a variety of structure prediction tasks. The biological accuracy of our lowest-energy structure predictions were verified to agree with Kato, et al. [23], whose scoring model we duplicate. Recall that the optimization problem is treated as a series of decision problems performing a binary search of the energy space. For each RNA sequence, a corresponding SAT instance is therefore constructed containing the energy and structural constraints along with an energetic bound that captures the minimum and maximum allowed energy for that step in the binary search. Given the precision of our energy model a search depth of 10 sufficed to identify the minimum energy structure of any structure tested.

4.2 Experimental Methodology

We solve the structure prediction problem using the following three methods:

- **Baseline Approach using CryptoMiniSat (BA):** A standard encoding of our problem in SAT. We generate the complete SAT encoding (with XOR clauses as appropriate) of the RNA secondary structure prediction problem, then use CryptoMiniSat to solve this problem. We also used MiniSat2 [11], and found that for this problem its performance is similar to CryptoMiniSat [26].
- **Offline Abstraction Refinement (OFFA):** An encoding of our problem using established refinement techniques. Starting with only the energy constraints from the SAT encoding of the RNA structure prediction problem to form the abstracted constraint, we use offline abstraction refinement to obtain a solution to the complete structure prediction problem. Each refinement step uses CryptoMiniSat to solve the current SAT problem, computes the set of constraints from the complete structure prediction problem that are inconsistent with this solution, and generates a new problem by incrementally adding these constraints to the

current problem in SAT. The refinement process continues until it produces a solution to the complete input problem.

– **Online Abstraction Refinement (ONA):** The methodology enabled by our tool Lynx. Starting with only the energy constraints from the SAT encoding of the RNA structure prediction problem to form the abstracted constraint, we use online abstraction refinement to obtain a solution to the complete structure prediction problem. After each CryptoMiniSat propagation step, the constraint manager examines the current partial solution to find the set of constraints from the full structure prediction problem that conflict with the current solution. It then incrementally adds these constraints to the current problem before CryptoMiniSat takes the next partial solution step. The difference between the Offline (OFFA) and Online (ONA) approaches is the granularity of the refinement steps. Each refinement step in the OFFA version takes place only after CryptoMiniSat produces a complete solution to the current problem. Each refinement step in the ONA version, in contrast, takes place at the much finer granularity, every time CryptoMiniSat extends the current partial solution.

RNA	sequence	Baseline	Offline	Online
	length	(sec)	Tot(sec)=SAT+Ref (# steps)	Tot(sec)=SAT+Ref (# steps)
PKB115	24	**1.4**	**1.7** = 1.3+0.4 (205)	**0.8** = 0.6+0.2 (2,538)
PKB102	24	**1.3**	**1.0** = 0.7+0.3 (129)	**0.6** = 0.5+0.1 (1,766)
PKB119	24	**2.1**	**3.6** = 3.0+0.6 (266)	**1.6** = 1.3+0.3 (4,108)
PKB103	25	**3.1**	**6.6** = 5.4+1.2 (417)	**3.5** = 3.1+0.4 (6,191)
PKB123	26	**5.6**	**24.7** = 22.7+2.0 (597)	**7.4** = 6.8+0.6 (8,980)
PKB154	26	**2.5**	**3.8** = 3.2+0.6 (236)	**1.9** = 1.7+0.2 (4,070)
PKB152	26	**3.2**	**6.2** = 5.2+1.0 (255)	**2.3** = 2.0+0.3 (5,528)
PKB126	27	**4.0**	**6.6** = 5.5+1.1 (384)	**2.8** = 2.5+0.3 (5,874)
PKB124	29	**4.7**	**5.1** = 4.4+0.7 (262)	**2.3** = 2.1+0.2 (4,635)
PKB100	31	**11.0**	**52.3** = 49.4+2.9 (315)	**6.8** = 6.0+0.8 (11,890)
PKB105	32	**17.0**	**58.3** = 54.0+4.3 (1004)	**18.1** = 17.0+1.1 (16,817)
PKB118	33	**13.7**	**32.8** = 29.6+3.2 (591)	**8.2** = 7.4+0.8 (12,878)
PKB120	36	**36.1**	**571.1** = 560.6+10.5 (652)	**24.1** = 21.9+2.2 (26,370)
PKB065	46	**185.1**	**11,341.9** = 11,298.7+43.2 (1,344)	**112.7** = 108.1+4.6 (50,508)
PKB205	48	**388.6**	**T.O.**	**391.6** = 381.9+9.7 (72,922)
PKB147	51	**1,917.3**	**T.O.**	**1,087.9** = 1,067.2+20.7 (131,321)
PKB248	66	**T.O.**	**T.O.**	**T.O.**
PKB072	67	**5,352.6**	**T.O.**	**2,414.1** = 2,367.6+46.5 (286,881)

Table 1. Comparison of running times between Baseline (BA), Offline (OFFA), and Online (ONA) methods. Total cumulative time (across all solver instances during search) is reported, broken down by the amount of time spent in the SAT solver versus the amount of time spent in refinement. The number of refinement steps involved is also given. T.O. indicates that a timeout occured after 1hr of an individual SAT solver instance.

RNA	sequence length	Baseline Mem(MB) / Clauses	Offline Mem(MB) / Clauses	Online Mem(MB) / Clauses
PKB115	24	**5.0** / 3,223k	**5.0** / 94k	**72.1** / 82k
PKB102	24	**5.0** / 3,219k	**5.0** / 86k	**5.0** / 75k
PKB119	24	**5.0** / 3,240k	**5.0** / 130k	**5.0** / 104k
PKB103	25	**5.0** / 4,142k	**16.5** /174k	**5.0** / 136k
PKB123	26	**43.4** / 5,244k	**19.7** / 226k	**74.7** / 168k
PKB154	26	**5.0** / 5,204k	**5.0** / 128k	**5.0** / 106k
PKB152	26	**5.0** / 5,220k	**16.6** / 174k	**5.0** / 128k
PKB126	27	**72.1** / 6,544k	**74.5** / 171k	**5.0** / 129k
PKB124	29	**5.0** / 10,076k	**5.0** / 142k	**5.0** / 108k
PKB100	31	**90.5** / 16,937k	**23.9** / 376k	**90.0** / 231k
PKB105	32	**157.4** / 20,584k	**75.9** / 448k	**95.7** / 260k
PKB118	33	**131.9** / 24,870k	**23.2** / 355k	**22.8** / 227k
PKB120	36	**276.0** / 42,698k	**76.7** / 729k	**75.3** / 369k
PKB065	46	**1,011.8** / 196,236k	**150.6** / 341k	**122.9** / 595k
PKB205	48	**1,221.3** / 255,861k	**T.O.**	**145.0** / 808k
PKB147	51	**1,988.9** / 373,294k	**T.O.**	**188.7** / 1,322k
PKB248	66	**T.O.**	**T.O.**	**T.O.**
PKB072	67	**9,046.5** / 2,031,362k	**T.O.**	**313.1** / 2,652k

Table 2. Comparison of memory usage between Baseline (BA), Offline (OFFA), and Online (ONA) methods. Given is the maximum memory (in MB) required throughout all SAT solver instances, along with the sum of the total number of clauses (in thousands) both input and generated during refinement. T.O. indicates that a timeout occured after 1hr of an individual SAT solver instance.

4.3 Results

Table 1 presents the total execution times required for the different methods to solve the RNA structure prediction problems. We run each method with a timeout of 3600 seconds for each SAT solution problem (i.e., each binary search step). Each row in the table corresponds to a single RNA. The first column is the number of base pairs in the RNA sequence. The next column presents the time (in seconds) required for the BA method to solve the problem. Recall that each problem requires the solution of 10 SAT instances; the reported total time is the sum of the 10 individual SAT solution times. The next column presents data from the OFFA method and is of the form $t = s + c(r)$. Here t is the total time required to solve the structure prediction problem (the sum of the solution times for the 10 SAT problems), s is the amount of time spent in the SAT solver, c is the amount of time spent in the constraint manager, and r is the total number of refinement steps (summed over all 10 SAT problems). The last column presents data from the ONA method and is also of the form $t = s + c(r)$.

Up to problem PKB124, the solution times for all of the methods are roughly comparable (all less than ten seconds and within a factor of two for the same RNA sequence). For larger problems the OFFA approach starts to exhibit substantially larger solution times than either BA or ONA approaches; for the largest problems in our benchmark set OFFA times out. For two of the largest three problem sizes BA is roughly a factor of two slower than ONA; BA times out for PKB248.

We note that there is a substantial difference between the number of refinement steps that the ONA and OFFA methods perform — OFFA typically performs hundreds of (relatively coarse grain) refinement steps, while ONA performs thousands of (fine grain) refinement steps. These data indicate that, as expected, the SAT solver can respond much more quickly to fine grain than to coarse grain refinement steps, but that the ONA method requires more fine grain steps to reach a solution.

Table 2 presents the maximum amount of memory required to solve the structure prediction problem (this is the maximum over all runs of the SAT solver of the amount of memory that the SAT solver consumes) and the total number of clauses for each RNA. For the OFFA and ONA methods, the total number of clauses is the sum over all binary search steps of the number of clauses in the problem at the final refinement step. Each entry of the table is in the form m/c, where m is the maximum memory and c is the number of clauses. Both the OFFA and ONA methods generate problems with substantially smaller numbers of clauses than the BA method (BA typically generates hundred to thousand times as many clauses OFFA and ONA typically generate). For larger RNA sequences, these larger clause sizes translate into substantially larger memory requirements for the BA method — OFFA and ONA never go above several hundred Mbytes, while BA starts requiring more than 1Gbyte of memory for the larger sequences.

4.4 Discussion

These data highlight how the ONA method is able to combine the benefit of small memory requirements, which it shares with OFFA, and feasible execution times, which it shares with BA (further note that ONA often exhibits roughly a factor of two performance advantage over BA). We attribute these characteristics to, first, the ability of the ONA method to effectively find relatively small problems whose solution also happens to be a solution of the complete structure prediction problem, and second, the ability of the ONA method to efficiently guide the SAT solver to the solution through fine-grain corrections to partial solution missteps. A comparison with the OFFA method illustrates how quickly correcting any missteps on the part of the SAT solver (by operating the refinement steps after every intermediate SAT solver decision rather than after every complete solution) can deliver very efficient solution times even in situations where the more coarse OFFA approach fails to solve the problem in an acceptable amount of time.

5 Related Work

There has been a lot of recent work on incremental SAT solvers [21], DPLL(T) [13], abstraction-refinement based techniques in the context of model-checking and decision procedures for SMT theories [2]. We summarize the related work, and contrast other tools with Lynx.

Incrementality and SAT Solvers: The work that is closest to ours is by Stuckey et al. [21] and the related idea of DPLL(T) [13]. Our work is different from Stuckey et al. in the mechanism employed to implement incrementality, namely, a callback interface. Our approach is more flexible in the sense that it can be used to expose other internals of SAT solvers (e.g., branching heuristics or restart triggers) to lay non-expert users in ways that makes the solver easy-to-experiment with and easily adaptable to different problem domains. While DPLL(T) is a very powerful idea, it is also heavy-weight in the requirements placed on user-code (to ensure completeness and soundness) and is probably best used by experts. Another difference is that in DPLL(T) the T-part is typically a first-order theory, whereas the user code in Lynx refers only to Boolean variables.

Abstraction-refinement in decision procedures: The idea of counter-example guided abstraction refinement was originally developed in the context of model-checking [7]. Since then the basic idea has been adapted in different ways to solve the satisfiability problems of SMT theories [2]. Kroening, Ouaknine, Seshia, and Strichman [16] were the first to adapt CEGAR to deciding quantifier-free Presburger arithmetic. The same group then extended the idea to deciding the theory bit-vectors [6]. More recently, Brummayer and Biere give a new technique that allows early termination of an under-approximation refinement loop even when the original formula is unsatisfiable [5]. Ganesh and Dill proposed the use of abstraction-refinement for deciding the theory of arrays [12]. Gershman and Strichman extends the idea of abstraction-refinement to a heuristic for clause ordering in Boolean SAT solving [14].

RNA secondary structure prediction: Zuker introduced the first optimal algorithms for RNA secondary structure prediction based on a dynamic programming solution to energy minimization [28], although many improved predictors have followed [18]. Non-thermodynamic approaches have also met success through the use of phylogenetic relationships [15], or via machine learning [10]. The first efficient thermodynamic-based algorithm for predicting RNA pseudoknotted secondary structure was introduced by Rivas and Eddy (PKNOTS [25]). Subsequent algorithms have recognized alternate classes of pseudknots or improved upon the efficiency of solutions [20, 4], including the IP formulation focused on in this paper [23], and heuristics such as HotKnots [24].

6 Conclusions

We present Lynx, a programmatic incremental SAT solver that allows non-expert users to easily introduce domain-specific or instance-specific code into modern CDCL SAT solvers, thus enabling users to control the behavior of the solver in ways not possible otherwise. While there has been previous work on incremental SAT [21] and related ideas such as DPLL(T), Lynx's interface is simple to use and the requirements placed on user code are very minimal. The approach is a template on how to expose other internals of the SAT solver to non-expert users in a easy-to-use and intuitive way. We demonstrate the benefits of Lynx through a first-of-its-kind solver case-study from computational biology, namely, RNA secondary structure prediction.

References

1. PseudoBase RNA sequence website: http://pseudobaseplusplus.utep.edu/. Most widely used database for research on RNA sequences with Psuedoknots.
2. SMTLIB website: http://combination.cs.uiowa.edu/smtlib/.
3. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Feb. 2009.
4. M. Bon, G. Vernizzi, H. Orland, and A. Zee. Topological classification of RNA structures. *J. Mol. Biol.*, 379(4):900–911, 2008.
5. R. Brummayer and A. Biere. Effective bit-width and under-approximation. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *EUROCAST*, volume 5717 of *LNCS*, pages 304–311. Springer, 2009.
6. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS 2007*, volume 4424 of *LNCS*, pages 358–372, 2007.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 2003.

8. A. Condon, B. Davy, B. Rastegari, S. Chao, and F. Tarrant. Classifyin rna pseudoknotted structures. *Theoretical Computer Science*, 320:35–50, 2004.
9. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.
10. C. Do, D. Woods, and S. Batzoglou. CONTRAfold: RNA secondary structure prediction without energy-based models. *Bioinformatics*, 22(14):e90–e98, 2006.
11. N. Een and N. Sorensson. An extensible SAT-solver. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing*, pages 78–92, May 2003.
12. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*, pages 519–531. Springer-Verlag, 2007.
13. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures, 2004.
14. R. Gershman and O. Strichman. HaifaSat: A new robust SAT solver. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Haifa Verification Conference*, volume 3875 of *LNCS*, pages 76–89. Springer, 2005.
15. B. Knudsen and J. Hein. RNA secondary structure prediction using stochasatic context-free grammars and evolutionary history. *Bioinformatics*, 15:446–454, 1999.
16. D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *In: Proc. CAV. Volume 3114 of LNCS. (2004)*, pages 308–320. Springer, 2004.
17. R. Lyngso and C. Pedersen. Pseudoknots in RNA secondary structures. In *RECOMB'00*, pages 201–209, April 2000.
18. D. Mathews, M. Disney, J. Childs, S. Schroeder, M. Zuker, and D. Turner. Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure. *Proc. Natl. Acad. Sci.*, 101:7287–7292, 2004.
19. D. H. Mathews, J. Sabina, M. Zuker, and D. H. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *Journal of Molecular Biology*, 288(5):911–940, 1999.
20. D. H. Mathews and D. H. Turner. Prediction of RNA secondary structure by free energy. *Curr. Opin. Struct. Biol.*, 16:270–278, 2006.
21. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation = lazy clause generation. In C. Bessiere, editor, *CP'07*, volume 4741, pages 544–558. Springer-Verlag.
22. M. Parisien and F. Major. The MC-Fold and MC-Sym pipeline infers RNA structure from sequence data. *Nature*, 452:51–55, 2008.
23. U. Poolsap, Y. Kato, and T. Akutsu. Prediction of RNA secondary structure with pseudoknots using integer programming. *BMC Bioinformatics*, 10:S38, 2009.
24. J. Ren, B. Rastegari, A. Condon, and H. H. Hoos. HotKnots: Heuristic prediction of RNA secondary structures including pseudoknots. *RNA*, 11:1494–1504, 2005.
25. E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *J. Mol. Biol.*, 285:2053–2068, 1999.
26. M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT'09*, volume 5584 of *LNCS*, pages 244–257. Springer.
27. D. W. Staple and S. E. Butcher. Pseudoknots: RNA structures with diverse functions. *PLoS Biol.*, 3(6):e213, 2005.
28. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981.